DIGITECH

Simone Guggiari
Zusammenfassung 2015

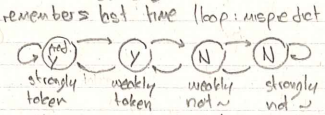
acle XOR minimal

23. ADVANCED MICROPROCESSORS

tricks invented to increase perform.

deep pipelining: increase #stages to about 10-20
limited by: hazards, seq. overhead, power (E), heat), $

branch prediction static: if back, assume yes
dynamic: keep history of last 100's of branches and their outcome
1 bit: remembers last time (loop: mispredict first & last)
2 bit:



superscalar processor multiple copies of datapath (multiple instr per clk)
difficulty: dependencies

out of order processor looks ahead across multiple instr. to issue as many as possible at once (if no dependencies)
problems: RAW, WAR, WAW } always when write (content is changed)

SIMD one instruction on array of data → used alot in graphics

multithreading process: program running on computer
thread: part of program (each process has many)

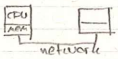
CONTEXT SWITCHING: pause, save - load, continue

MULTITHREAD: } throughput increases

multiprocessors: more communicating cores

SMT (symmetric multiprocessing)

asymmetric clusters



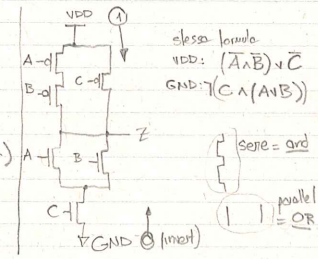
parallelism & reg renaming } known techniques

history: 1995 → 500 nm, 4 mio trans
2015 → 28 nm, 17000 mio "

24. REVIEW

ABSTRACTION LEVEL	EXAMPLE
Application SW	program
OS	device driver
Architecture	instr, reg
microarchitecture	datapath, controller
logic	adders, memories
Digital circuits	gates (AND, NOT, ..)
Analog circuits	Amplifiers
Devices	transistors
Physics	Diodes

1. INTRO



classic formula
VDD: $(\bar{A} \wedge \bar{B}) \vee \bar{C}$
GND: $\neg(C \wedge (A \vee B))$

2. BINARY NUMBERS

converter

0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111	
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Hex
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Dec

Bin → Dec: $1101_2 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 11_{10}$
Bin → Hex: $0100\,1110_2 = 4E_{16}$ use converter
Hex → Bin: $5A_{16} = 0101\,1010$ "
Hex → Dec: $3A6_{16} = 3 \cdot 16^2 + 10 \cdot 16^1 + 6 \cdot 16^0 = 934$
Dec → Bin:

193	1
96	0
48	0
24	0
12	0
6	0
3	1
1	1
0	

$193_{10} = 11000001_2$

Dec → Hex

500 : 16 = 31 r 4	31 : 16 = 1 r 15	1 : 16 = 0 r 1
-48	-16	-0
20	(15)	(1)
-16	$15_{10} = F_{16}$	$1_{10} = 1_{16}$
(4)	(Terne)	(contrario)
$4_{10} = 4_{16}$		
(units)		

$500_{10} = 1F4_{16}$ ⚠ ser interessato ai resti

signed bin. number $0101_2 = +5_{10}$ $1101_2 = -5_{10}$
sign magnitude
⊖ problems with addition: doesn't take account overflow
⊖ $-6 + 6 \neq 0$
range $[-(2^{n-1}-1), 2^{n-1}-1]$

One's complement invert 1 numer per negativ $+5 = 0101$, $-5 = 1010$
⊖ 2 zero representation
→ overflow: make overflow to LSB
10000 (for 4 bit)
range: same as, es 8bit: $[-127, 127]$

Two's complement invert bits, add 1 for negative (same way −→+)
$+5 = 0101$
$-5 = 1010 + 1 = 1011$
⊕ single zero representation
⊕ if overflow, it is simply ignored
⊕ associativity laws hold
⊕ standard bin. addition works without extra effort
range $[-2^{n-1}, 2^{n-1}-1]$
es 8 bit $[-128, 127]$

unsigned consider only positive range $[0, 2^N - 1]$

sign extension: sign bit is copied into all new MSB

Powers of 2:

0	1	2	3	4	5	6	7	8	9	10	M	12	13	2^N
1	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	

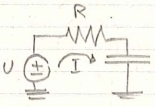
$2^{10} \approx 1'000$ = kilo $2^{20} \approx 1'000'000 = 1$ mio = mega $2^{30} \approx 1$ mrd = giga

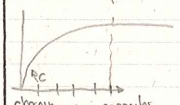
3. ELECTRICAL ENGINEERING

goal of circuit design is to optimize
- area (directly prop. to cost)
- speed/throughput (work faster, do more)
- power/(E) (eg. mobile devices, low performance = low(E))
- design time (expensive, vs. competition)

electric relations

$U = R \cdot I$
Volt = Ω · Ampère





charging a capacitor takes time (5 times = >99%)

$C = \frac{Q}{U}$

$I = \frac{dQ}{dt}$

Resistor: impedisce il passaggio di elettroni (lo rende più difficile)

Capacitor: (2 piastre vicinissime che non toccano) → fa aumentare il voltaggio fino a quando uguaglia con la corrente

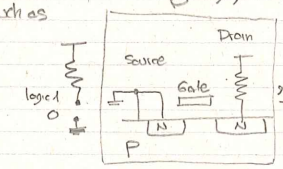
Building blocks for microchips: conductors: metals, aluminum, copper
insulators: glass, air
semiconductors: silicon, germanium

Semiconductor

N-type doping

P-type doping

By combining N and P regions next to each other on a pure semicond, you can create interesting things such as
- diodes, transistors, ...



mbare a se chiudi/apri il gate, la corrente passa/non passa indica 1/0

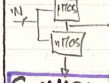
pMOS is the dual of nMOS

pMOS + nMOS = CMOS

what's so good about it:
- no input current
- no current when output is at logic level
- electrical properties determined by geometry
- simple to manufacture

CMOS gate structure

general law used to create any inverting logic gate (NAND, NOR, ..)



transistor
- parallel: on if either one on
- serie: all are on

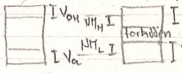
AND: NOT:
XNOR: (using only NAND)

Common logic gates:

Buffer	AND	OR	XOR	
NOT (inverter)	NAND	NOR	XNOR	

the static discipline given logically valid input, every circuit elem must produce logically valid output.

noise margins



$NMH = V_{OH} - V_{IH}$
$NML = V_{OL} - V_{IL}$

Moore's law: #transistors that can be manufactured doubles every 18 months

Power consumption: $P_{dynamic} = \frac{1}{2} C V_{DD}^2 f$ | $P_{static} = I_{DD} V_{DD}$ | $P_{tot} = P_S + P_D$

4&5. COMBINATIONAL CIRCUITS & (theory & design)

logic circuit composed of
- inputs/outputs
- functional specifications (relations between in/out)
- timing " (delay between ")

elements + wires

C: memoryless, out depends by current input

Rules
- every circuit elem itself is C
- every node of circuit is input, or connects to exactly 1 output
- no cyclic paths

bool rules

$BC = CB$
$(BC)D = B(CD)$
$B(C+D) = (BC)+(BD)$
$B(B+C) = B$
$(BC)+(B\bar{C}) = B$
$(BC)+(\bar{B}D)+(CD) = BC+\bar{B}D$
$B_0 B_1 B_2 \cdots = \bar{B}_0 + \bar{B}_1 + \bar{B}_2 \cdots$

$B+C = C+B$
$(B+C)+D = B+(C+D)$
$B+(CD) = (B+C)\cdot(B+D)$
$B+(BC) = B$
$(B+C)\cdot(B+\bar{C}) = B$
$(B+C)\cdot(\bar{B}+D)\cdot(C+D) = (B+C)(\bar{B}+D)$
$B_0 + B_1 + B_2 + \cdots = \bar{B}_0 \cdot \bar{B}_1 \cdot \bar{B}_2 \cdots$

Bubble pushing

pushing bubbles backward or forward changes the body of gate (AND ↔ OR)

(usually begin at output, go towards in)

Def: complement $\bar{A}$
literal: $A, \bar{A}$ (variable & complement)
implicant: product of literals $(A \cdot B \cdot \bar{C})$, $(\bar{B} \cdot C)$
minterm: product that includes all variables $(A \cdot B \cdot \bar{C})$
maxterm: sum " " $(A + \bar{B} + C)$

SUM of PRODUCTS (SOP)

A	B	Y	minterm
0	0	0	$\bar{A}\bar{B}$
0	1	1	$\bar{A}B$
1	0	1	$A\bar{B}$
1	1	1	AB

formed by ORing minterms for which output is TRUE (use this if few 1's)

$Y = (\bar{A}B) + (AB)$

each minterm is TRUE for the ROW

PRODUCT of SUMS (POS)

A	B	Y	maxterm
0	0	0	$A+B$
0	1	1	$A+\bar{B}$
1	0	0	$\bar{A}+B$
1	1	1	$\bar{A}+\bar{B}$

formed by ANDing the maxterms for which out is FALSE (Use this if few 0's)

$Y = (A+B)\cdot(\bar{A}+B)$ each maxterm is false for ro

Karnaugh Maps

notes:
- special order for bit comparison (00, 01, 11, 10)
- every 1 must be circled at least once
- each circle must spawn a power of 2
- " must be as large as possible
- a circle may wrap around the edges of the map
- a don't care X is circled only if it helps minimize the eq.

AB \ CD	00	01	11	10
00	1	0	X	1
01	0	X	X	1
11	1	1	X	X
10	X	1	X	X

$Y = A + C + \bar{B}\bar{D}$

X: 0 and 1 simultaneously, any value (≠ don't care, it's convention)
Z: not driving to any value (float), disconnected

Multiplexer selects one input

, it's a lookup table

does the opposite

Decoder

Glitch single input changes, multiple bits the output. doesn't cause problems, thanks to synchronous design conventions, but it's better not to have (result temporarily incorrect)

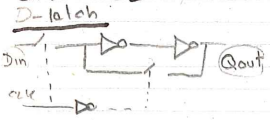
the black part fixes the glitch

8. Sequential Logic (circuits)

circuits that remember. It's everything that's not
output depends on current and prior input
state: all info about circuit to determine next behaviour

LATCHES

D-latch

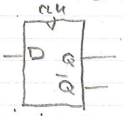


switch allows to change value
CLK controls it
only 1 switch active at time
- latch mode (opaque)
  loop active, keep state
- transparent mode
  input propagates

it's a bistable circuit

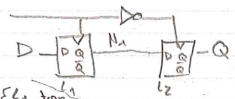
(requires FB)

FLIP FLOP

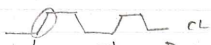


samples D on rising edge
otherwise Q keeps value
→ this is edge triggered value.

2 backtoback latches

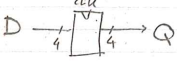


CLK=0 {L1 trans / L2 opaq
CLK=1 {L1 opaq / L2 trans



↳ on posedge, D propagates through Q

REGISTER set of flip flops to store multiple bits

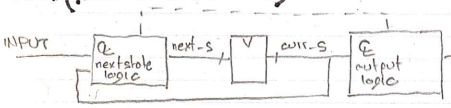


Design rules:
- break cycles using registers synchronized to CLK (state of system)
- every elem is comb circuit or register
- at least 1 reg (all with same CLK)

result: flip flop doesn't always sample → stability
- enabled: switch, if ON accepts new data
- resettable: forced to be 0
  ↳ synchronous: resets on CLK
  asynch: resets right away
- settable: forced to 1

FB

FSM (finite state machine)



state reg
FB (store & load)

k registers:
can be in 2^k states
(finite #)

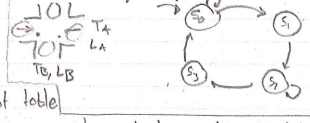
steps: 1) advance state
2) calculate next state
3) calculate output

Moore: default
Mealy: output depends on input as well
Medvedev (Moore with no out logic circuit)

Factoring FSM:
→ break complex FSM into simpler interacting FSM
(output of some is input of others)

Design: identify IN-OUTs
sketch state-transitions
write table
select state encodings
[Moore: finish state-trans table
write output table
[Mealy: rewrite combined state trans + output table
write bool equations
sketch circuit scheme

es. traffic lights



FSM state transition table

curr state	inputs	next state
S_{00}	0 x	
S_{01}	1 x	
...		

FSM output table

curr state	output

one hot encoding (1 state bit/state)
es. 4 states
0001
0010
0100
1000
} always 1 active
this makes next state & output logic simpler

FSM encoded state trans. table
(some, but use bits)

9. VERILOG FOR FB

module: main building block
defines name input/output
to instantiate it:
safe way, declare what connects to what

if you do single bit op on busses
(es &) it will treat each bit singularly

ternary operator: assign y = (bool) ? yes:no;

```
module example (input a,
                input[3:0] b,
                input c,d,
                output y
                );
endmodule
```

example ex_1(.a(A), .b(B), ...)
original wire in this file

express numbers

N'b xxx... -xx
↓ ↘
#bits base

b	binary	2
h	hexadecimal	16
d	decimal	10
o	octal	8

can also include X (don't care)
Z (both)

operators

~	not
* / %	mult, div, mod
+ -	add, sub
<< >>	shift (logic)
<<< >>>	" (arithm)
< <= => >	compare
== !=	eq, neq
& ~&	AND, NAND
^ ~^	XOR, XNOR
| ~|	OR, NOR
? :	ternary op

HIGH ↑ precedence ↓ LOW

always @(sensitivity list) //posedge, negedge
contains ↳ when this happens, code in block is executed
FB q <= d; // assign
↳ has to be declared as reg !! // FB

Synchronous: things happen only with CLK
Asynchronous: " whenever we define it (es reset, prepolente, subito)

use begin as {
end }

you can have multiple always blocks

assignment is different within always block

non-blocking

```
always @ (*)
begin
  a <= 2'b01;
  b <= a;
end
```

values are assigned at the end of the block, all together (parallel)
→ will get same result as blocking, but with more iterations

blocking

```
always @ (*)
begin
  a = 2'b01;
  b = a;
end
```

more like programming, sequential execution is interrupted (blocked) to assign value

rules: always @ (posedge CLK) <= // non blocking } SYNC FB
always @ (*) = // blocking } complex FB in which you use if/else/case/...

⚠ do not assign multiple times to same variable in always block!

assign y = a&b; } continuous assignment for simple FB

FB - blocks with memory (FF, latches, FSM)
- triggered by CLK (latches → sensitive to 0/1
FF → pos/negedge)

⚠ if in the always we define results for all possibilities, it becomes FB, so always is not useful (no need to remember)

es. always @ (inv, data)
if (inv) result <= ~data;
else result <= data;

you can use if/else/then/case (= switch)

```
case (data)
  0: ___ ;
  1: ___ ;
  default: ___ ; //use it !!
```

⚠ becomes FB if you define everything

casez: checks for "don't care"

FSM in verilog

```
parameter S0 = 2'b00; // param optional, better reading
reg [1:0] state, nextstate

//state register
always @ (posedge clk, posedge reset)
  if (reset) state <= 0;
  else       state <= nextstate;

//next state logic
always @ (*)
  case (state)
    S0: nextstate = S1;
    S1: ___
    default: ___
  end case

//output logic
assign y = (state == S0);
```

Remember: always used for elements that have memory, or complex FB

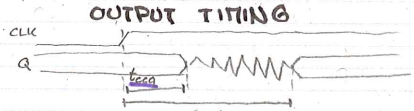
10. FB TIMING

flip flop samples @ posedge clk,
D must be stable otherwise METASTABILITY

INPUT TIMING

time before clk that has to be stable
time after
aperture time

OUTPUT TIMING

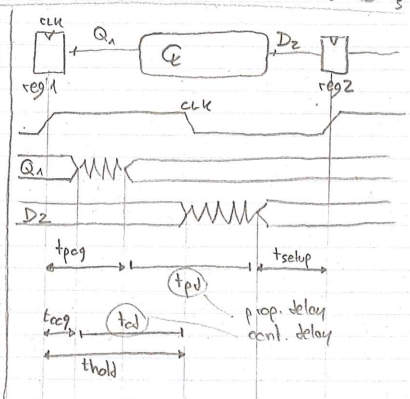


cont. min. delay: time after clk that might be unstable (min delay in-out)
prop. delay: time after clk that signal is guaranteed to be stable (max delay in-out)

delay between registers has min and max delay, dependent on delays of circuit elements
CLK period (T_c) = time between 2 posedge
frequency: $f_c = \frac{1}{T_c}$ more freq = more work done

1 MHz = 10^6 1/s
1 GHz = 10^9 1/s

time constraints



setup: depends on max delay from reg1 through FB
input to reg2 must be stable at least t_{setup} before

$$T_c \geq t_{pcq} + t_{pd} + t_{setup} + \bullet$$

$$t_{pd} \leq T_c - (t_{pcq} + t_{setup})$$

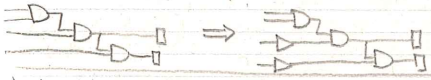
• = skew
• = uncertainty setup

hold: depends on min delay from reg1 through FB
the input to reg1 must be stable at least t_{hold} after

$$t_{hold} + \bullet < t_{ccq} + t_{cd}$$

$$t_{cd} > t_{hold} - t_{ccq}$$

fixing time violations → add buffers to short paths to slow them down.



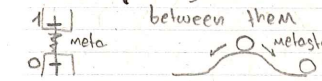
Clock skew: clk doesn't arrive everywhere at same time
skew = difference between 2 posedge
↳ examine worst case to ensure no violation!

SETUP: worst if CLK2 reaches than CLK1 → rallenta (freq possibile clk
HOLD: later

see formulas in previous page

Dynamic discipline: might be violated by async input (es. user)

Metastability: bi-stable device has 2 stable states and metastability between them



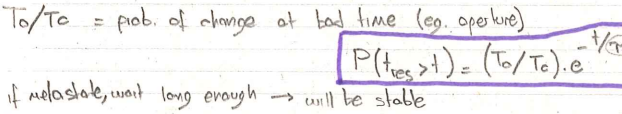
T_o/T_c = prob. of change at bad time (eg. aperture)

$$P(t_{res} > t) = (T_o/T_c) \cdot e^{-t/\tau}$$

→ const of FF (speed of leaving metastable)

if metastable, wait long enough → will be stable

Synchronizer: used to synchronize unreliable async inputs
↳ makes prob. of failure small (but not 0)



back-to-back Flipflops

$(T_c - T_{setup})$ time to get stable

Probab. of failure

$$P(fail) = \frac{T_o}{T_c} \cdot e^{-\frac{T_c - t_{setup}}{\tau}}$$

$$P^*(fail) = N \cdot P(fail)$$

if change is 1/s

if ——— N/s

it fails once every $\frac{1}{P^*(fail)/sec}$

$$MTBF = \frac{1}{P^*(fail)/sec} = \frac{T_c \cdot e^{\frac{T_c - t_{setup}}{\tau}}}{N \cdot T_o}$$

↳ mean time between fail

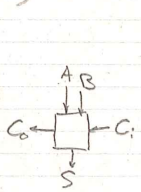
11. ARITHMETIC CIRCUITS

- hearth of digital system (determin speed/area/clk)
- focus for improvement
- part that does something other than copying data

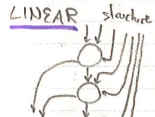
CAN DO
shift / rotate
compare incr/decrement negate add/sub
division sqrt multipl
exp log/trigonometric

Half adder

Full adder
(all 3 single bit #)

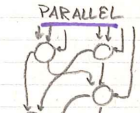


LINEAR structure



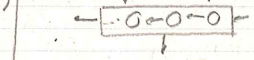
4.TO

PARALLEL (tree struct)

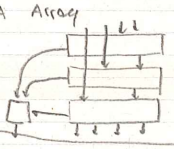


3T.U

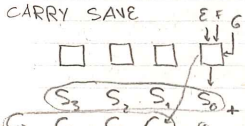
RIPPLE CARRY ADDER (RCA)



RCA Array



CARRY SAVE



result (1 time unit delay)

$E_1 F_1 G_1 = S_1 + C$

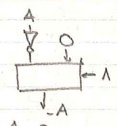
multipliers
3 steps:
- partial product generation
- carry save tree to reduce partial products
- CPA (carry prop. adder) to finalize

binary mult:

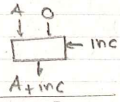
```
1 1 0 1 x
0 1 0 1
-------
1 1 0 1
0 0 0 0
1 1 0 1
0 0 0 0
-------
1 0 0 0 0 0 1
```

Deriv ates

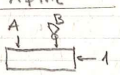
NEGATOR (2's complement)
$-A = \bar{A} + 1$



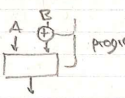
INCREMENTER
subtractor (-inc)



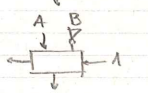
SUBTRACTOR
b is inverted (+1)



ADD/SUB



COMPARATOR



= EQ
!= $\overline{EQ}$
> GE $\overline{EQ}$
>= GE
< $\overline{GE}$
<= $\overline{GE}+EQ$

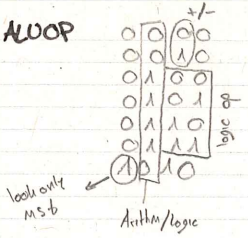
Arithm. op without adders

Shifters: logical (fills with 0)
11001 » 2 = 00110
arithmetic (on right, fills with MSB)
11001 »ₐ 2 = 11110

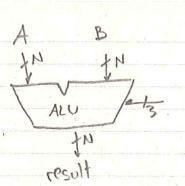
Rotator: loop
11001 ROR 2 = 01110
ROL

shamt
A → » → result
« multipl by 2^n
»» division

ALUOP



ADD
SUB
& and
| or
^ xor
~| nor
SLT set less than



12. Number SYSTEM

- how to represent floats, irrgular

int: unit (1)
sign (+/-)
2's compl

fractions: fix point 110101.001
float point $1.1011 \cdot 2^6$

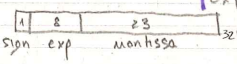
-7.5 to fix point 8 bit, 4.4
7.5 = 0111.1000
invert 1000 0111
+1 1000 1000

float, like scientific notation

$\pm M \cdot B^e$

mantissa
base
exponent

first bit of mantissa always 1 (implic)



representations:
1) standard
2) leave out implicit 1 of mantissa
3) IEEE: use biased exponent → 8bit bias = 127 = 01111111
bias + exp = saved #
es 7 → 134 = 10000110

single precision:
bias = $2^7 - 1 = 127$
double precision:
$2^{10} - 1 = 1023$ (hold it in 2 32-bit registers)

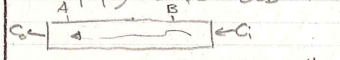
addition is complex:
- extract + add 1 to mant
- compare exp - shift mantissa
- add mantissas
- normalize & round, adjust mantissa
- assemble back

Special #:

	sign	exp	fraction
0	x	0	0
∞	0	1	0
-∞	1	1	0
NAN	x	1	X (not 0)

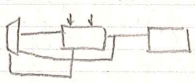
13. FAST ADDERS

RCA: cause of the carry to speed up for better performance
propagates from LSB to MSB (crit path is rare, but still)



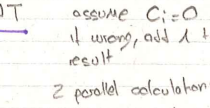
DIVIDE LARGE ADDER (4 methods)

- **SKIP**



carry can be
- propagated, if it comes out of old
- generated here, if it doesn't

- **GENERATE**: if (A&B) $C_o = 1$ (always)
- **PROPAGATE**: if (A^B) $C_o = C_i$ (dependent)

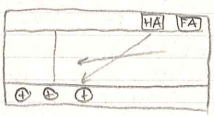


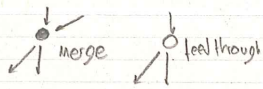
- **INCREMENT** assume $C_i = 0$ if wrong, add 1 to result
- **SELECT** 2 parallel calculations (assume 0 and 1) – select right one
- **LOOK AHEAD** precompute carries

$C_0 = C_i$
$C_1 = G_0 + P_0 C_i$
$C_2 = G_1 + P_1 G_0 + P_1 P_0 C_i$
...

PPA (parallel prefix adders) quite complex, not seen in depth

- pre-processing
- carry (the only difference)
- post-processing





this create the dotted schema that makes up the carry grid

performance: # black dots (Area)
stages (critical path)
max fan out (stage delay)

- SK Sklolansky
- CIA carry inc adder
- BK brent kung
- KS kogge stone
- HC Hoan Carlsson

in verilog, simply write
assign a = b+c
compiler decides best architecture

14. TESTBENCH

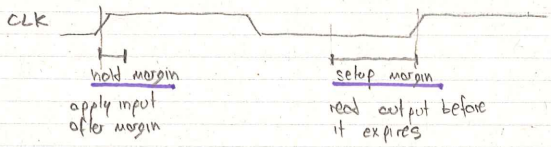
check if circuit works (give input and see if output matches the expected)
DUT: device under test (UUT, unit)

types: simple – just feed input and return result (you check)
self-check – like simple, but also looks at output and compares with "right solution"
self-check with vectors
$display to print.
⚠ you make same amount of errors here

using vectors: the more elaborate
write testvector file: input and expected output
(use high level / golden model to generate correct output)

Testbench steps:
1. generate clk for assigning/reading input/outputs
2. read test v. in array
3. assign in, read outputs computed
4. compare results against vector (out and expected out) report errors

⚠ brute force is not feasible. Use formal verification, test critical cases only



#10 (wait 10 ns) → never change at clk, blurry signal sometimes

$readmemb ("example.tv", testvectors); // read from file into array

%h //print in hexadecimal

15. MIPS ASSEMBLY

(microprocessor without interlocked pipeline stages)

architecture: programmer's view (instructions and registers)
micro arch: implementation
instructions: word that means something } assembly machine
instr. set: vocabulary

John Hennessy Inventor

principles:
- simplicity favours regularity
- make the common case fast
- small is fast
- good design means compromises

RISC (reduced instr. set computer) has few op, like MIPS
CISC (complex ———) es. INTEL: lots of instr
⊖ too power consumption
⊕ faster
less instructions
⊕ saves power (good for mobiles)
× ARM:

add a, b, c // a = b+c

add rd, rs, rt parse, always 2 sources, 1 destination

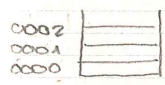
only simple instr have operators (complex can use more operators)

operands: where to find data.
- register (fast) ⟶ 32 reg of 32 bit each; small = fast
- memory (big) commonly used → kept in reg
- constant (immediate)

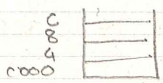
0.	$0 = 0		28	$gp	glob. pointer
1.	$at	assemble temp	29	$sp	stack point
2-3	$v	procedure return	30	$fp	frame pointer
4-7	$a	procedure args	31	$ra	return ad.
8-15, 24-5	$t	temp			
16-23	$s	saved			
26-7	$k	OS temp			

immediates:
16 bit, 2's compl. constant
es lw, addi a = b+4
memory or reg address not required (directly available)

Word address mem
each 32 bit data word has address (also of 32 bit)



Byte address mem
each 8 bit byte has address (words address increments by 4)

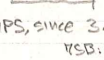


MIPS uses this!

Read: lw Write: sw

lw $s3 1($0)
m source (1+0)

sw $t4 0x7($0)
from to
reg mem address

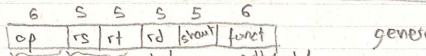
Little endian: start at LSB


(MIPS, since 31:0) MSB:LSB

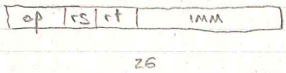
Big endian: start at MSB

doesn't matter which is used, as long as they don't share

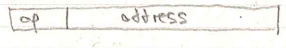
TYPES

R-type


general instr.

I-type


es lw, sw memory & immediate

J-type


branch / jump

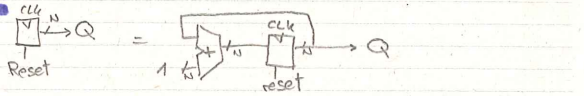
⚠ order (rt, rs) - tag, source lw rt i(rs)
sw "

32 bit instr & data, all stored in mem. (text editor, video game, ...) all sequence of instr.

PC (program counter) = address of next op
- fetch instr
- execute
- repeat with next

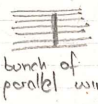
MIPS starts @ 0x00400000

16. MEMORIES

counters:


shift reg

load: tells if it has to copy data in next FF

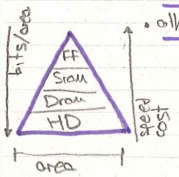


Serial / Parallel conversion

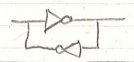
IN	OUT	
S	S	→ shift reg
S	P	
P	S	
P	P	→ normal reg

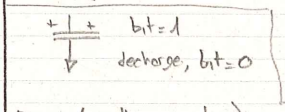
Memory: large part in circuit. very useful, reprogrammable

methods:
- flip flop (very fast, parallel, expensive) 20 trans.
- static RAM (fast, sequential, less exp) 6 "
- dynamic RAM (slower, volatile*, sequential, cheap) 1 "
- other (HD, flash, ...) (much slower, non volatile) cheapest (no transistors directly in oxide)



volatile: loses data on read and without power (S+D ram)
non ": doesn't (es ROM, read only mem)


Sram (2 inverters, similar to D-latch)



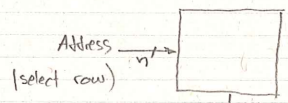
per cambiarlo, se colleghi transistor con più carica, puoi convincerlo più facilmente

(let it loop, finally the big data will prevale)

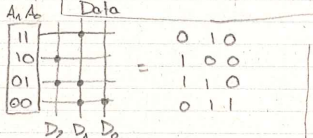
Dram (small capacitor)
⚠ slowly loses charge → has to be refreshed
the longer it is, (more charge), the longer it stays

ROM: much denser, non volatile (no need for transistors for value change)
es. OS, configuration data, lookup tables (log, trig, ...)

Array memory 2D cells



ROM logic



lookup stored result

$D_2 = A_1 \wedge A_0$
$D_1 = \overline{A_1} + A_0$
$D_0 = \overline{A_1}\,\overline{A_0}$

⊕ = or
⊙ = and

reg [2:0] RAM [15:0]

Port = address/data pair
3 ported: 2 read RD_1, RD_2
1 write WD

WE = write enabled (clk always checks it)

17. MICROARCHITECTURE

how to implement architecture in hardware

there is:
- single cycle (raw) → 1 instr = 1 clk
- multicycle → = more, faster clk
- pipelined → like multicycle, but more instr executed together

Data mem: store bunch of data

control signals:
- ALU inputs
- WA write address
- WE write enable ← mem reg
- WD write data

| | | | | |

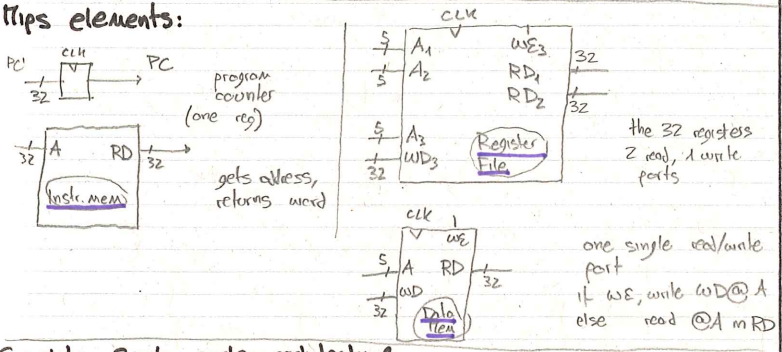
Branches: control flow of PC

beq: if (rs == rt) $pc = BTA → branch target address: = PC+4 + (imm << 2)

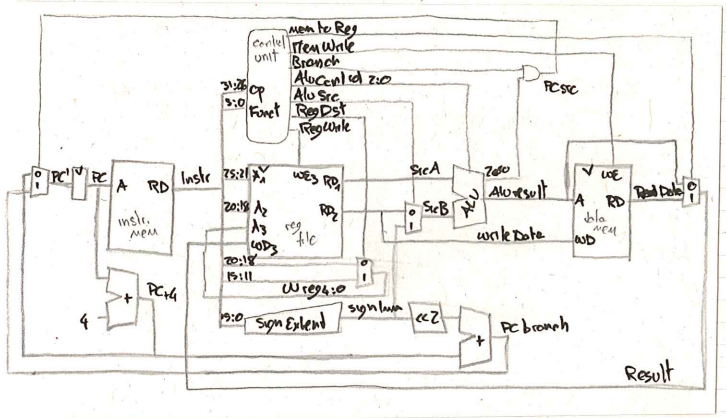
bne: branch if not equal

jump: no "if", simply jump — jal (jump and link) saves current PC+4 in $ra
jr (jump register) use immediate

Mips elements:



Complete Single cycle architecture



Performance

cycles per instruction = CPI (1 or more)
1 clock cycle time = CLK period (critical path)
1/CLK period = CLK frequency

we have N instructions
f = max frequency → 1/f = clock time, period

TIME of program: $N \times CPI \times \frac{1}{f}$

to improve:
- ⊖ # instruction N
  - make them do more
  - better compiler
- ⊖ CPI
  - simpler instr (RISC)
  - multiple ALU/units/cores
- ⊕ frequency
  - new technology
  - redesign crit. path
  - pipeline

timing

T_c → time of CLK is limited by critical path (slowest)

$$T_c = t_{PCQ_PC} + t_{MEM} + \max\{t_{RF\,read}, t_{setup}, t_{MUX}\} + t_{ALU} + t_{MEM} + t_{MUX} + t_{RF\,setup}$$

in most implementations, mem/alu/reg file are the slowest

ps (10^{-12} s)
ns (10^{-9} s)
μs (10^{-6} s)

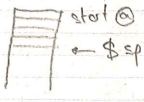
18. MIPS PROGRAMMING

(for these examples, see directly code on slides)

- Arrays: large amount of similar data
- Procedures: caller → calling procedure (gives args, jumps to it)
  callee → called (does procedure, returns result, sets PC back to caller)

use non var declaration registri, metti elem in array e ritorna indirizzo 1° elem.
- must not overwrite important info
  use stack to save temp. variables overwritten. the stack automatically expands/contracts
  LIFO

→ high to low address

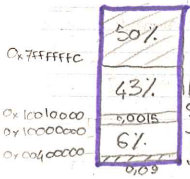


Registers

reserved	free to use
$s (save on stack if want to use)	$t
$ra, $sp	$a
stack above $sp	$v

if you nest multiple calls, you have to save $ra aswell

Mips memory map



Code:

beq $s0, $s1, target

#if branch is taken, this is not executed

target:
add ...

j $s0 # jump to PC = $s0

if (a==b) bne $, $, L1
else ...
j done
L1: ...
done:

while ($!= $) while: beq $, $, done ... j while done:

for (i=0; i!=10; i++) # init
for: beq $, $, done ... addi $, 1 j for done:

add $a, $b, $c	a=b+c
addi $a, $b, 10	a=b+k
and $a, $b, $c	a=b&c
sll $a, $b, 2	a=b<<2
lw $a, 100($b)	$a=mem[]
sw $a, 100($b)	mem[]=$a
beq $a, $b, lab	jump to lab if
bne $a, $b, lab	
slt $a, $b $c	if(b<c) a=1 else a=0
j lab	
jr $ra	(procedure return)
jal lab	(procedure call)

es. <10
for: slt $t1, $s0, $s1
beq $t1, $0, done
...
addi $, 1
j for
done:

Addressing modes

- reg only
- immediate
- base address 72($0)
- PC-relative = usavi 3 right bit del numero PC
- pseudo-direct

dato che non puoi salvare 32 bit in 26, salva quanto devi aggiungere (poco prob. devi andare fuori alla parte della memoria)

pseudo instructions:
have more native MIPS instructions underlying
es mul $, $, $ = mult $, $
mflo $

Compile & run

high level code
↓ compiler
assembly code
↓ assembler
object file
↓ linker (libraries)
executable
↓ loader
memory

Exceptions

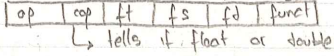
when one happens, processor
→ records what caused it
→ jumps to default exception handler
→ returns. done

es: arithm. overflow, hardware interrupt, breakpoint, 0 division, ...

SIGNED op: add / addi / sub
UNSIGNED op: addu / addiu / subu

floating point op: there is a register file with 32 float values ($f0 - $f31) double uses 2 reg

F-type:

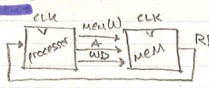


↳ tells if float or double precision

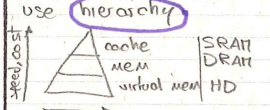
19. CACHE ARCHITECTURE

performance depends on
- processor speed
- memory system speed

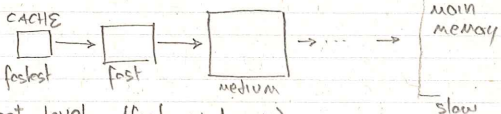
but memory hasn't gotten as fast as processor : create model that is!



use hierarchy



Multilevel



CACHE = highest level (fast, ~1 CLK)
ideally, it has data required by processor
→ anticipate needs (but impossible to predict 100%)
→ use locality! • copy recent data + nearby data (block of size M)

Locality: use it for fast memory
TEMPORAL → time, data seen recently likely to be seen again
SPATIAL → space, data close to data seen likely to be seen next

therefore keep recent data at front, and nearby data also in high levels of mem

memory: hit — record found in this cache level
miss — " not, must go into next

hitrate = # hits / # mem accesses (HR)
missrate = # miss / " (MR)

AMAT: Average mem. access time (for 1 access)

$$AMAT = t_{cache} + MR_{cache}(t_{mem_1} + MR_{mem_1}(t_{VM}))$$

VM: virtual mem
→ you can chain on more lvl

Terminology

- capacity — # bytes
- block size — # bytes brought to cache together
- N of blocks
- associativity degree — # blocks in set
- N of sets

C	
b	
B = C/b	
N	
S = B/N	

Address 11...111 111 00 — sempre 0 (multiplo di 4)

bits that determine position in cache (both set, block offset)

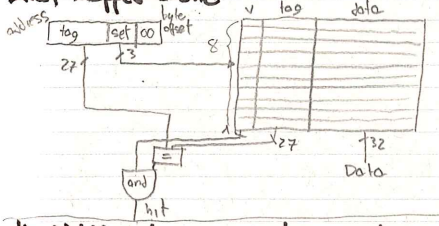
CACHE

is divided in sets

address maps to only 1

- direct mapped (1 block/set)
- N way set associative (N blocks/set)
- Fully associative (all blocks / 1 set)

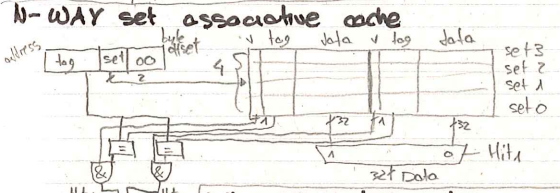
Direct mapped cache



in tag we save the address (ID) of the data found in the cell, so we know whether the data is what we are looking for.

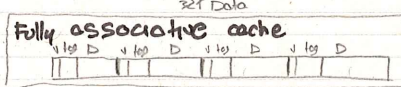
v is a bit that tells if it's stored something (easy to reset)

N-WAY set associative cache



this system reduces conflict misses

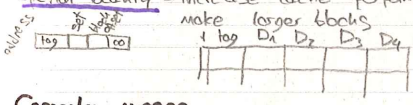
Fully associative cache



very expensive

Spatial locality — increase cache performance

make larger blocks



Capacity misses

If cache is full, to add new element one is removed (eg. 4). If we access it again, we have capacity miss.

SOLUTION: use bigger cache

usually, 4 = LRU (least recently used)

In direct mapped, x can go only 1 place

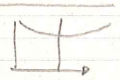
N-associative, ——— multiple places (N)

↳ this reduces conflicts misses

Types: (misses)

- compulsory (1st access)
- capacity
- conflict (maps to same loc)
- miss penalty: time to get data from lower level

keep block size reasonable for max gain (both extremes hurt performance)



Virtual mem

paragraph skipped, not in exam

20. I/O SYSTEMS

interface peripherals

mem. mapped I/O

interact with environment! (eg. buttons...)

accessing peripherals is like new areas

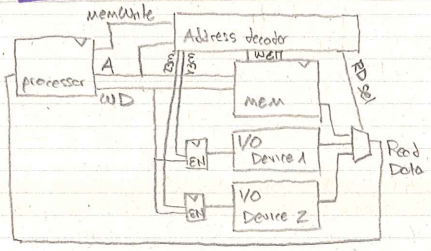
→ each device has set (range) of addresses

OS: allocate device (says syllabe based on input, won't finish pronounce, says next)

device driver is software to allow interfacing and fetching values from I/O Device

NEEDED:

- address decoder — see what (mem/device) is communicating
- I/O registers — holds values
- RD multiplexer — selects between mem/device

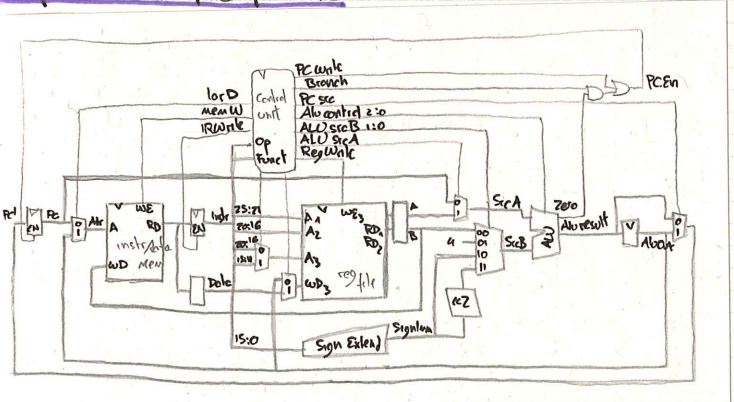


21. MULTICYCLE MIPS PROCESSOR

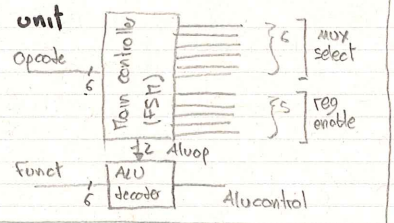
single cycle	vs	multi cycle (more realistic)
⊕ simple		⊕ higher CLK
⊖ cycle determined by slowest instr.		⊕ simple instr run fast
⊖ 2 ALU's & 2 mem (1 mem instr, 1 mem data)		⊕ reuse hardware
		⊕ smaller (thus cheaper)
		⊖ sequencing overhead
		⊖ more complex control unit
		⊖ not always faster

we want single ALU & mem

Complete multi-cycle processor



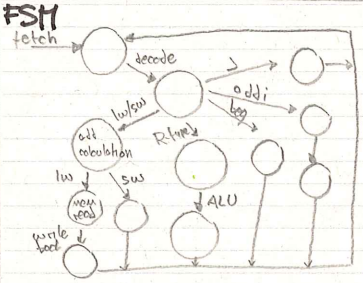
control unit



es. lw

1. read instr
2. read register
3. calculate address
4. read content @ add.
5. write back

FSM



each ○ is a state, takes 1 clk

instructions take ≠ # of cycles

3 clk	beq, J
4 clk	R-type, sw, addi
5 clk	lw

CPI is average, es 25% load, 10% store, 11% branch, 2% jump, 52% R-type → CPI ≈ 4.12

Performance

sequencing overhead: multicycle might be slower than single!

- not all steps same length (time waste)
- seq. overhead! (tpcq + tsetup = eg. 50ps)

multicycle critical path:

$$T_c = t_{pcq} + t_{MUX} + \max(t_{ALU} + t_{MUX}, t_{MEM}) + t_{setup}$$

22. PIPELINED MIPS ARCHITECTURE

Parallelism spatial → duplicate hardware, multiple tasks at once

temporal → task broken in stages. Pipelining (eg assembly line)

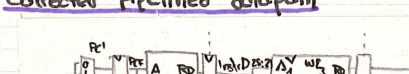
Def: token: group of input processed to get group output

latency: time for 1 token (start → finish) } we want ↓ lat, ↑ throughput

throughput: # tokens/unit of time } (eg. baked cookies)

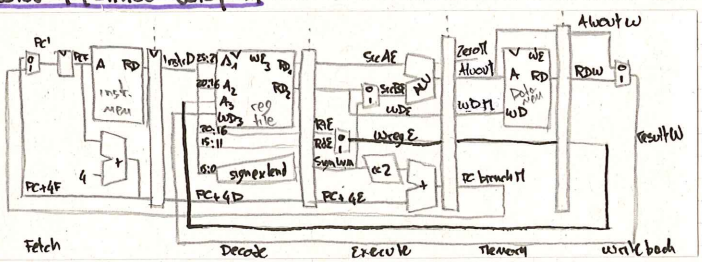
Pipelined processor (temporal parallelism)

break single cycle processor in stages:



- fetch instr
- decode (read reg)
- execute (ALU)
- memory (read/write)
- write back (reg)

Corrected pipelined datapath



Hazard: when instruction depends on result from previous instr not yet computed

Data hazard: — write done in first half of clk } save clk not a problem!

— read second

→ solutions:

- insert NOP (no operation) that does nothing } save clk cycles
- rearrange code if not dependent
- forward data using "Hazard Unit" (take data directly when ready from wire, don't wait for it to be in reg)
- stall the processor (es lw, very slow, 2 clk latency, can't use other techniques)

stalls supported by HW:

Control hazard: next instruction (PC) not decided yet (eg. branch)

branch not determined until 4th stage

→ assume 0/1 of branch, and load corresponding instructions

→ if mispredicted, flush and load correct (you lost several clk)

early branch resolution: determine/guess branch before it happens

branch prediction • backward branches usually taken (loops, more likely to stay in)

es!

suppose: • consider history of branch for extra accuracy

40% load used by next instr → CPI = 2 (ex 1)

25% branches mispredicted

average $CPI_{lw} = 0,6 \cdot 1 + 0,4 \cdot 2 = 1,4$

$CPI_{beq} = 0,75 \cdot 1 + 0,25 \cdot 2 = 1,25$

Pipeline performance: 5 stages

$T_c = \max$ {

fetch	$t_{pcq} + t_{mem} + t_{setup}$,
decode	$2(t_{RF_read} + t_{mux} + t_{eq} + t_{AND} + t_{mux} + t_{setup})$,
execute	$t_{pcq} + t_{mux} + t_{mux} + t_{ALU} + t_{setup}$,
memory	$t_{pcq} + t_{memwrite} + t_{setup}$,
write back	$2(t_{pcq} + t_{mux} + t_{RF\,write})$

}

slowest of determines CLK speed.

→ decode and write back do their thing in ½ clk, therefore they get 2*

performance summary

processor	performance	speedup
single-C	95 s	1
multi-C	133 s	0,71
pipelined	63 s	1,51 (unfortunately not 5x)